Speedrunning the Lakehouse

A composable FaaS over object storage

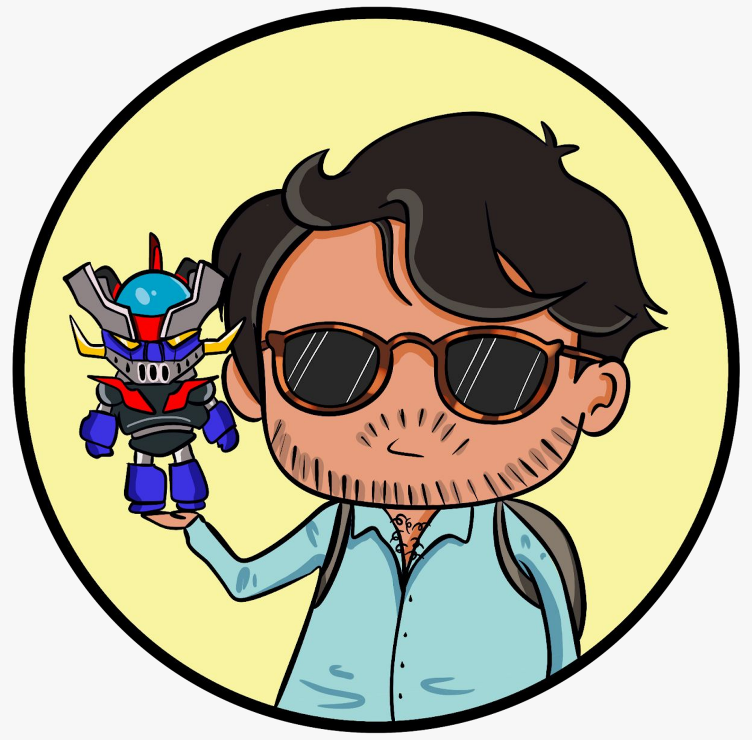
CDMS@VLDB
09.05.25

Ciao, I’m Jacopo!

- Co-founder and CTO at **Bauplan**.
  Backed by IE, SPC, Wes McKinney, Spencer Kimball, Chris Re *et al.*.
- Started the “Reasonable Scale” movement.
  Co-founder at Tooso and lead AI at TSX:CVO after the acquisition.
- 10 years up and down the stack in R&D, product, open source
  ICML, KDD, VLDB, NAACL, SIGIR, WWW *et al.*, >2k stars, >50M+ downloads.

It takes a (distributed) village

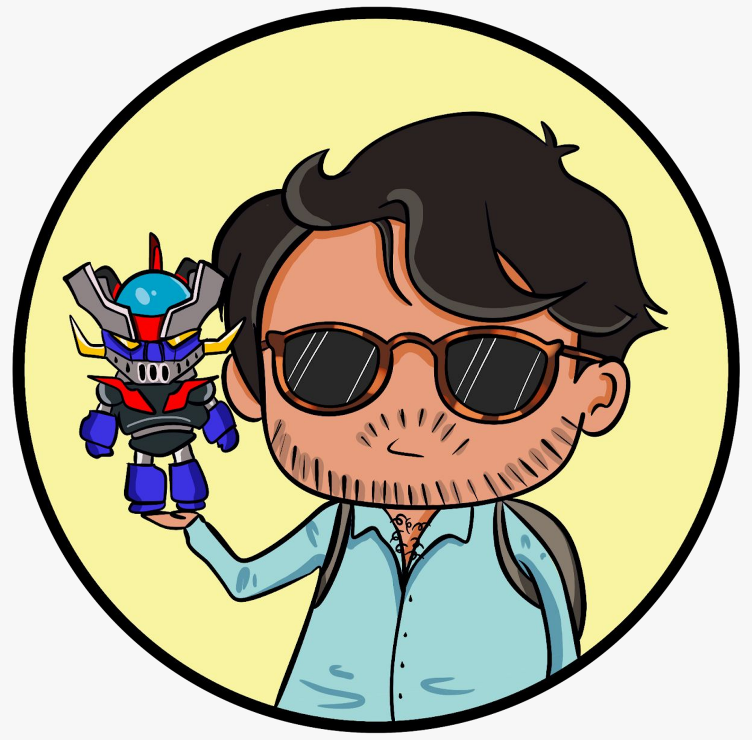
> Matt, Ciro, Luca, Nate, Vlad (and others, unfortunately without a chibi) share with me the credit for whatever value these ideas may have.


It takes a (distributed) village

> Matt, Ciro, Luca, Nate, Vlad (and others, unfortunately without a chibi) share with me the credit for whatever value these ideas may have.

> Obviously, all the remaining mistakes are theirs 😁



"bauplan is [a system] fully built using composable principles (...). It is refreshing to learn about a real-life system built using such architectural principles."

Reviewer #2

speed·run

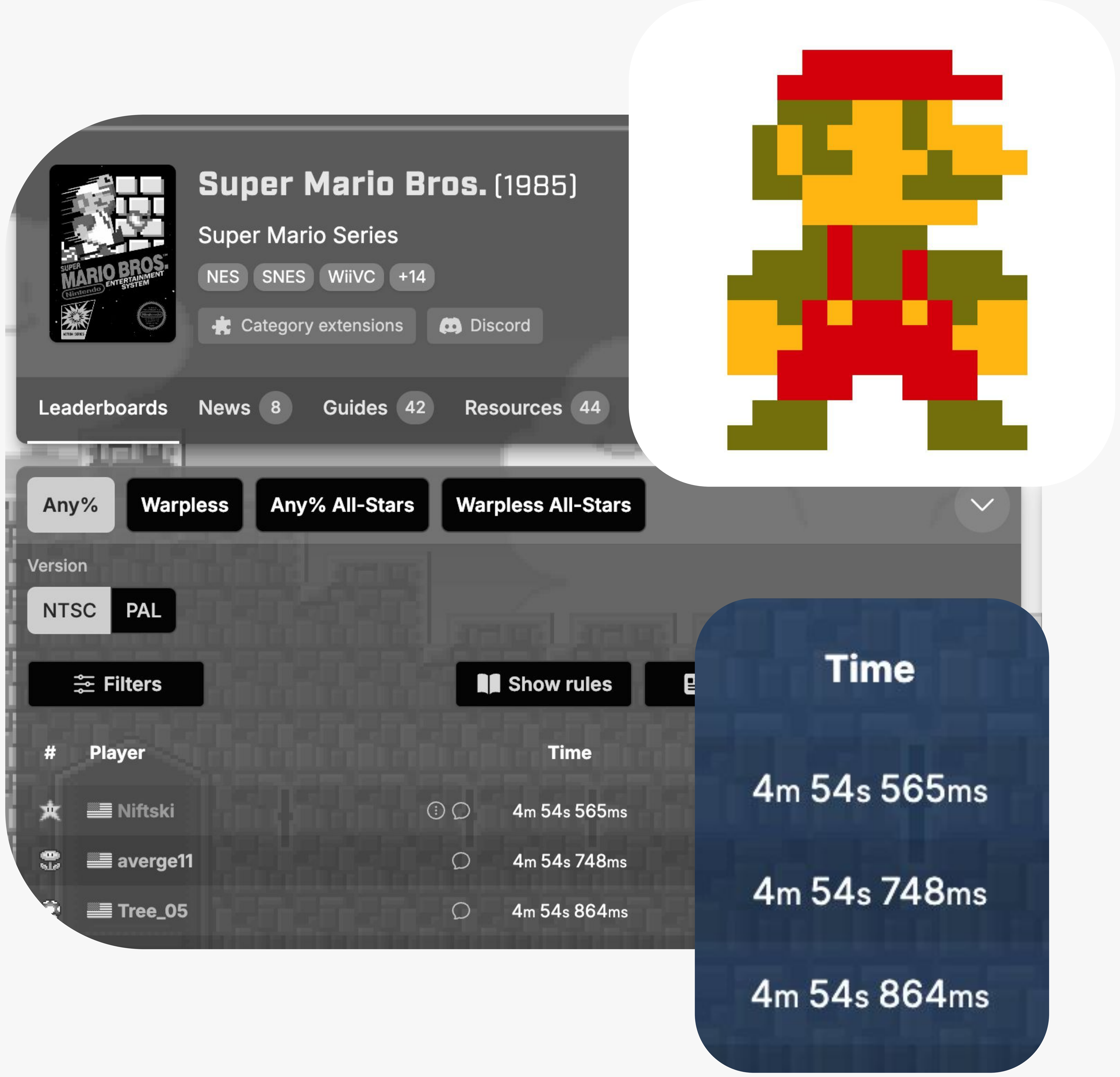
/ˈspēdˌrən/

verb

gerund or present participle: **speedrunning**

complete (a video game, or level of a game) as fast as possible.

"I used to be able to speedrun this game in less than 20 minutes"



Speedrunning a Lakehouse? Really?

Speedrunning a Lakehouse? Really?

1. Simplicity
2. Composability

4:17:01 (!?!)


YouTube
Search
Create
SPARK ARCHITECTURE
1TB
Driver program
SparkContext
Cluster Manager
Worker Node
Executor
Cache
Task
Task
W1
Worker Node
Executor
Cache
Task
Task
W2
19:38 / 4:17:01 [redacted]
[redacted] Tutorial (From Zero to Hero) [redacted] Masterclass
Ansh Lamba
46.6K subscribers
Join
Subscribe
4.8K
Share
Download
All
From Ansh Lamba
Big Data
All In One RMM Solution

Mo (mental) models, mo problems

Interaction	UX	Infrastructure
Traditional DLH		
Batch pipeline	Submit API	One-off cluster
Dev. pipeline	Notebook Session	Dev. cluster
Inter. query	Web Editor (JDBC Driver)	Warehouse

Eudoxia: a FaaS scheduling simulator for the composable lakehouse

Tapan Srivastava*
tapansriv@uchicago.edu
University of Chicago
Chicago, Illinois, USA

Jacopo Tagliabue*
jacopo.tagliabue@bauplanlabs.com
Bauplan Labs
New York, USA

Ciro Greco
ciro.greco@bauplanlabs.com
Bauplan Labs
New York, USA

ABSTRACT

Due to the variety of its target use cases and the large API surface area to cover, a data lakehouse (DLH) is a natural candidate for a composable data system. *Bauplan* is a composable DLH built on "spare data parts" and a unified Function-as-a-Service (FaaS) runtime for SQL queries and Python pipelines. While FaaS simplifies both building and using the system, it introduces novel challenges in scheduling and optimization of data workloads. In this work,

data lake and warehouse, such as cheap and durable foundation through object storage, compute decoupling, multi-language support, unified table semantics, and governance [19].

The breadth of DLH use cases makes it a natural target for the philosophy of composable data systems [23]. In this spirit, *Bauplan* is a DLH built from "spare parts" [31]: while presenting to users a unified API for assets and compute [30], the system is built from modularized components that reuse existing data tools through novel interfaces: e.g. Arrow fragments for differential caching [29],

```
pip install bauplan
bauplan checkout my-branch
bauplan run
```

“Simplex Sigillum Veri”

A sample pipeline

transactions

ID	USD	COUNTRY
13	44	US
144	13	IT
146	1	IT

```python
def euro_selection(
  df=transactions
):
  _df =
transform_input(df)
  return _df
```

euro_selection

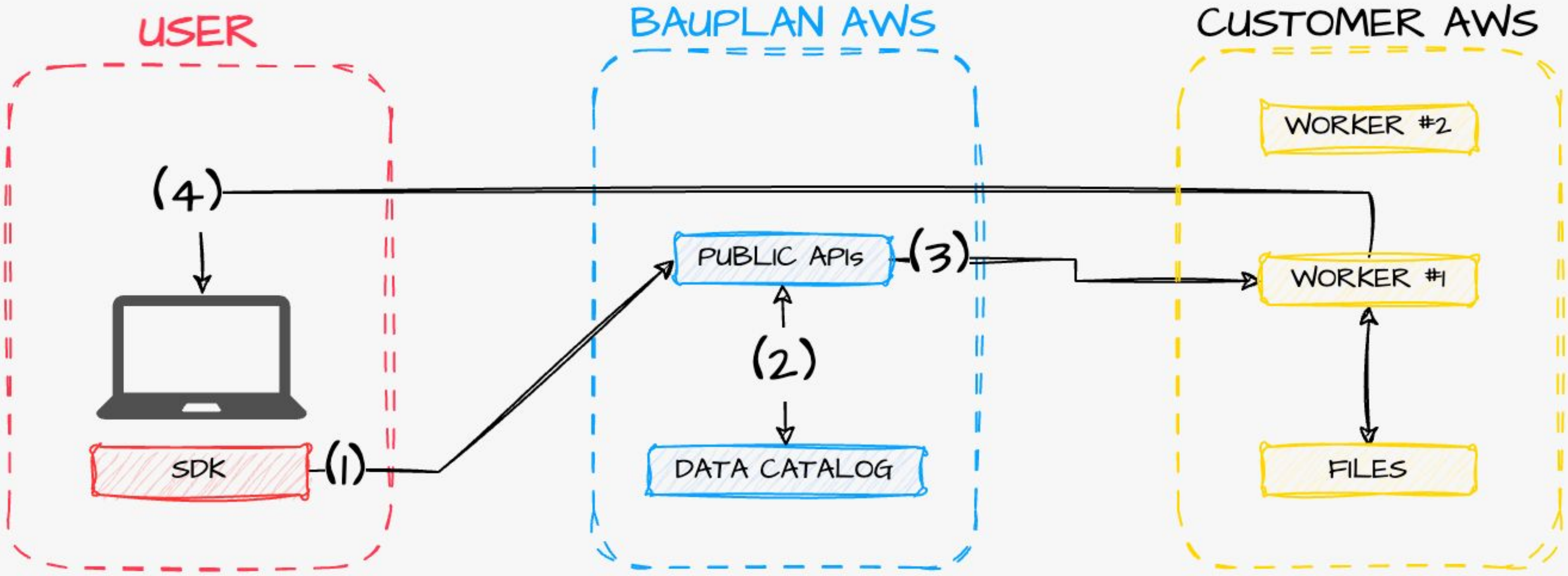
ID	USD	COUNTRY
144	13	IT
146	1	IT

```python
def usd_by_country(
  df=euro_selection
):
  _df =
transform_input(df)
  return _df
```

usd_by_country

COUNTRY	USD
IT	14

High-level view of a `bauplan run`



CLI / SDK

Control Plane

Data Plane

High-level view of a bauplan run

Pipelines are chained functions (Batch / Dev)

Queries are chained functions as well!

Everything is a function, or “OnlyFaas”

Easy to reason about

- Simple abstractions, “looks like code”
- A unified compute model, “everything is a function”

[VLDB 2023: Building a serverless Data Lakehouse from spare parts]()

PROs: one mental model to rule them all

Can we re-use existing FaaS? **NO!!!**

- Resource limitations
- No "DAG awareness"
- Slow feedback loop

Interaction	UX	Infrastructure
Traditional DLH		
Batch pipeline	Submit API	One-off cluster
Dev. pipeline	Notebook Session	Dev. cluster
Inter. query	Web Editor (JDBC Driver)	Warehouse

Middleware 2024: Bauplan: zero-copy, scale-up FaaS for data pipelines

CONs: we need a new a FaaS-for-data

- New programming model
  - Express data and code dependencies
- New runtime
  - Function lifecycle
  - Scheduling

Interaction	UX	Infrastructure
Traditional DLH		
Batch pipeline	Submit API	One-off cluster
Dev. pipeline	Notebook Session	Dev. cluster
Inter. query	Web Editor (JDBC Driver)	Warehouse

[Middleware 2024: Bauplan: zero-copy, scale-up FaaS for data pipelines]

New programming model

clear “division of labor” between platform and users

New programming model

bau.py

```python
@bauplan.model()
@bauplan.python(
    "3.11",
    pip={"polars": "1.33.0"}
)
def euro_selection(
    data=bauplan.Model(
        "transactions",
        columns=["id", "usd", "country"],
        filter="eventTime BETWEEN 2023-01-01 AND 
2023-02-01"
    )
):
    # filtering here
    # return a dataframe
    return _df
```

bau.py

```python
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"polars": "0.8.8"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

New programming model

User code here!

bau.py

```python
@bauplan.model()
@bauplan.python(
   "3.11",
   pip={"polars": "1.33.0"}
)
def euro_selection(
    data=bauplan.Model(
       "transactions",
       columns=["id", "usd", "country"],
       filter="eventTime BETWEEN 2023-01-01 AND 2023-02-01"
   )
):
    # filtering here
    # return a dataframe
    return _df
```

bau.py

```python
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"polars": "0.8.8"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

New programming model

Signature Table(s)->Table

bau.py

```python
@bauplan.model()
@bauplan.python(
   "3.11",
   pip={"polars": "1.33.0"}
)
def euro_selection(
    data=bauplan.Model(
       "transactions",
       columns=["id", "usd", "country"],
       filter="eventTime BETWEEN 2023-01-01 AND 2023-02-01"
   )
):
    # filtering here
    # return a dataframe
    return _df
```

bau.py

```python
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"polars": "0.8.8"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

New programming model

Infra-as-code

bau.py

```python
@bauplan.model()
@bauplan.python(
   "3.11",
   pip={"polars": "1.33.0"}
)
def euro_selection(
    data=bauplan.Model(
       "transactions",
       columns=["id", "usd", "country"],
       filter="eventTime BETWEEN 2023-01-01 AND
2023-02-01"
   )
):
    # filtering here
    # return a dataframe
    return _df
```

bau.py

```python
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"polars": "0.8.8"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

New programming model

I/O chaining

bau.py

```python
@bauplan.model()
@bauplan.python(
   "3.11",
   pip={"polars": "1.33.0"}
)
def euro_selection(
    data=bauplan.Model(
       "transactions",
       columns=["id", "usd", "country"],
       filter="eventTime BETWEEN 2023-01-01 AND
2023-02-01"
   )
):
    # filtering here
    # return a dataframe
    return _df
```

bau.py

```python
@bauplan.model(materialize=True)
@bauplan.python(
    "3.10",
    pip={"polars": "0.8.8"}
)
def usd_by_country(
    data=bauplan.Model("euro_selection")
):
    # aggregation here
    # return a dataframe
    return _df
```

New runtime

we can't just "run user functions", which is a challenge and opportunity bauplan run = (plan + environment + data movement)

Planning

USER CODE

bau.py

```python
@bauplan.model()
@bauplan.python(
    "3.11",
     pip={"polars": "0.8.8"}
)
def euro_selection(
    data=bauplan.Model(
        "transactions",
        columns=["id", "usd", "country"],
        filter="eventTime BETWEEN 2023-01-01 AND 
2023-02-01"
    )
):
    # filtering here
    # return a dataframe
    return _df
```

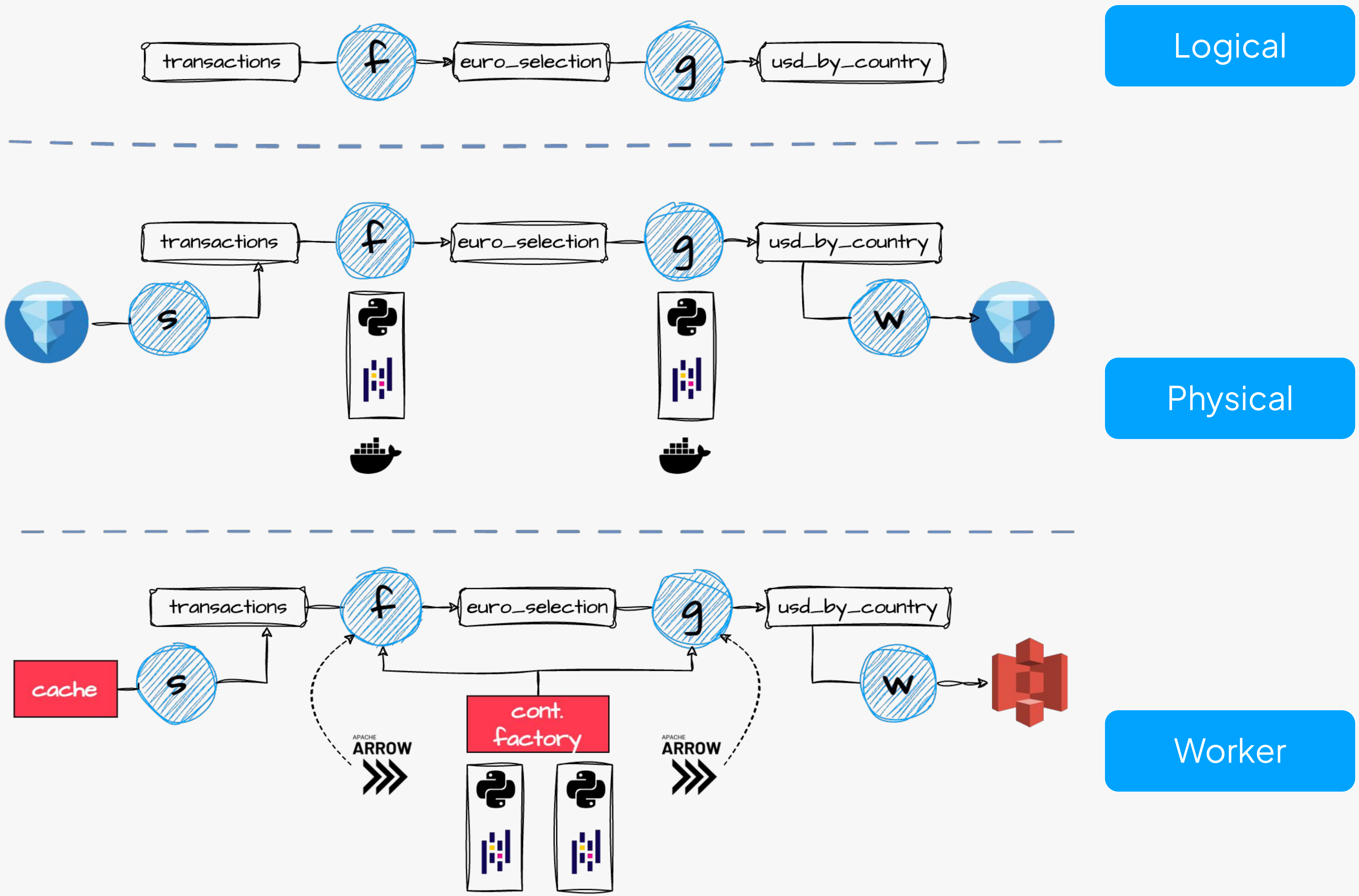
PLATFORM CODE

docker

```
RUN ...
…
```

```
obj.get(Range='bytes=32-64')['Body']
…
```

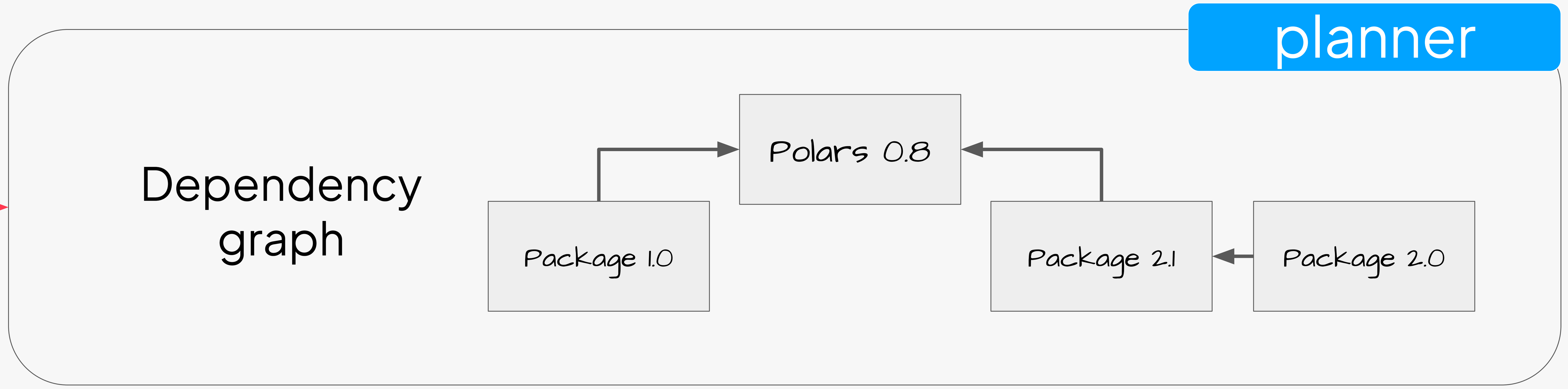
Planning



CDMS@VLDB2025: Dag lakehouse planning with an ephemeral and embedded graph database

Environment

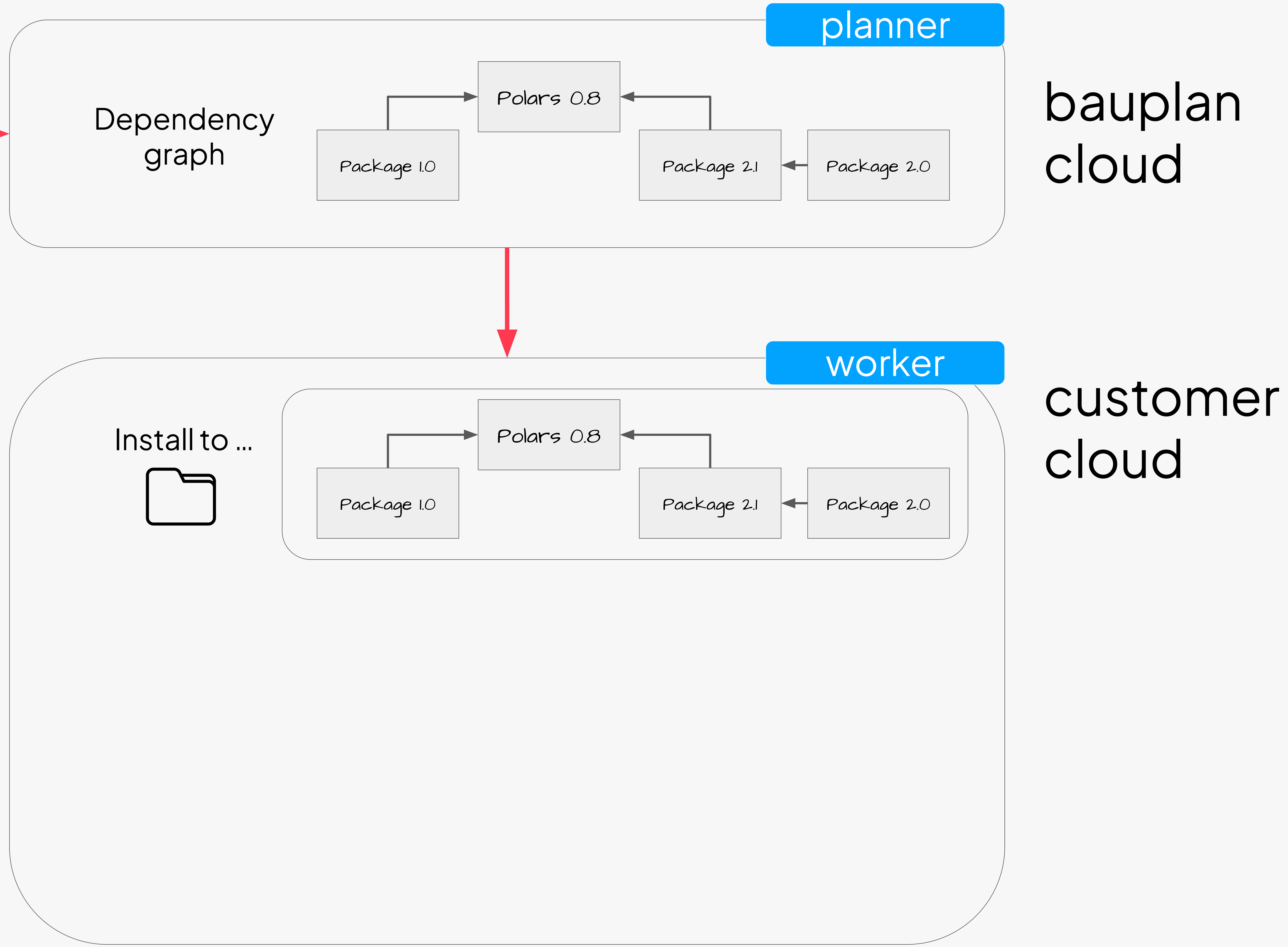
```
@bauplan.python(
    "3.11",
     pip={"polars": "0.8.8"}
)
```



bauplan cloud

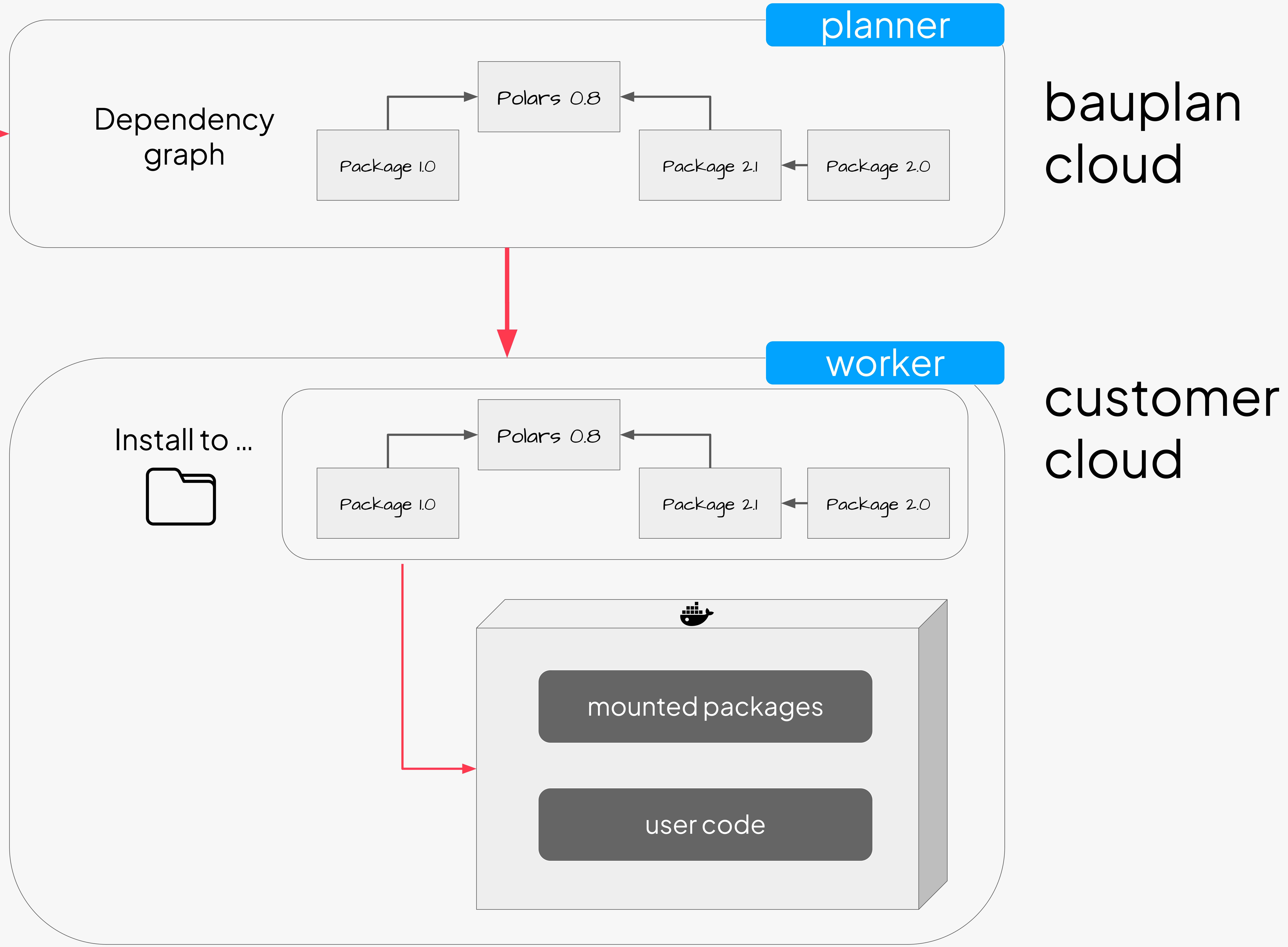
Environment

```
@bauplan.python(
    "3.11",
     pip={"polars": "0.8.8"}
)
```



Environment

```
@bauplan.python(
    "3.11",
     pip={"polars": "0.8.8"}
)
```



Environment: assemble, don't build

- **NO** Docker, **NO** bandwidth bottlenecks, **NO** ECR update
- Functions are ephemeral: no lifecycle management.
- **Adding a package is 15× faster than AWS Lambda**

Table 2: Time to add *Prophet* to a serverless DAG

Task	Seconds
AWS Lambda[4]	
Update ECR container and function	130 (80 + 50)
Snowpark	
Update Snowpark container	35
bauplan	
Update runtime	5 / 0 (cache)

Data movement: Arrow everywhere + zero-copy

```
data=bauplan.Model(
        "transactions",
        columns=["id", "usd", "country"],
        filter="..."
    )
```

- Across workers, an Arrow stream is as fast as local parquet files (**B**)
- Within a worker, tables can be zero-copy shared between functions (**C**)

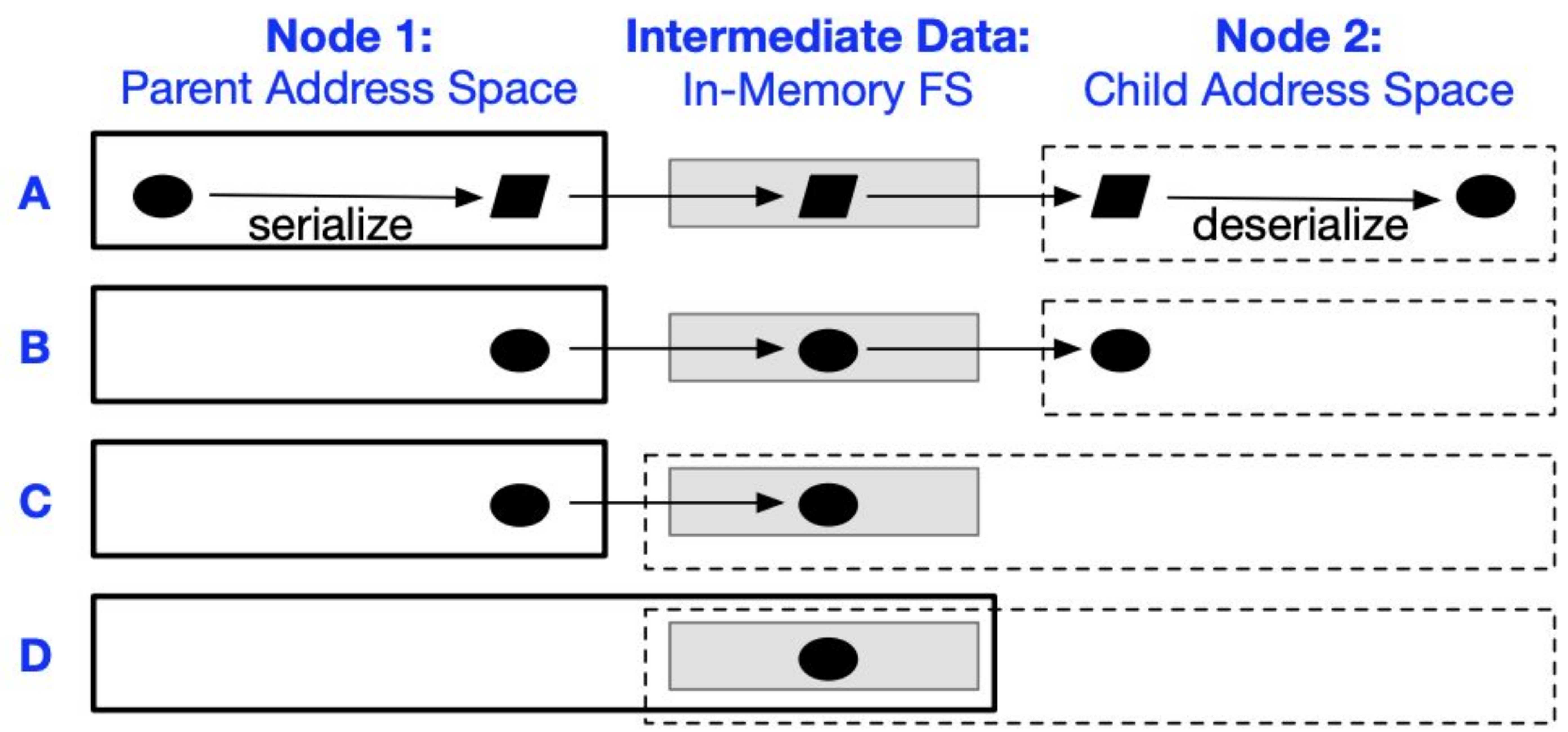


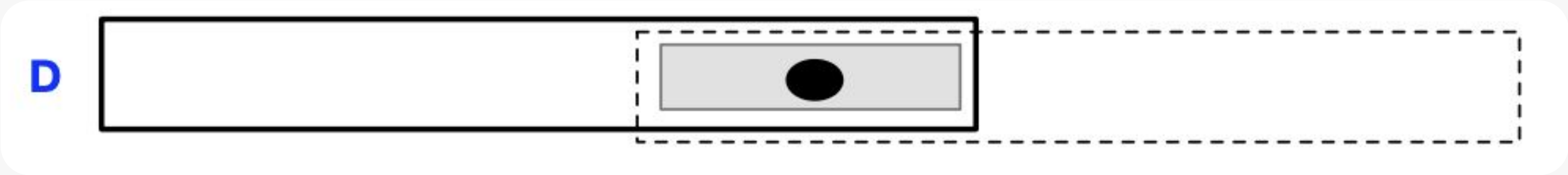
Figure 1: **Communication: Degrees of Zero Copy**

Data movement: Arrow everywhere + zero-copy

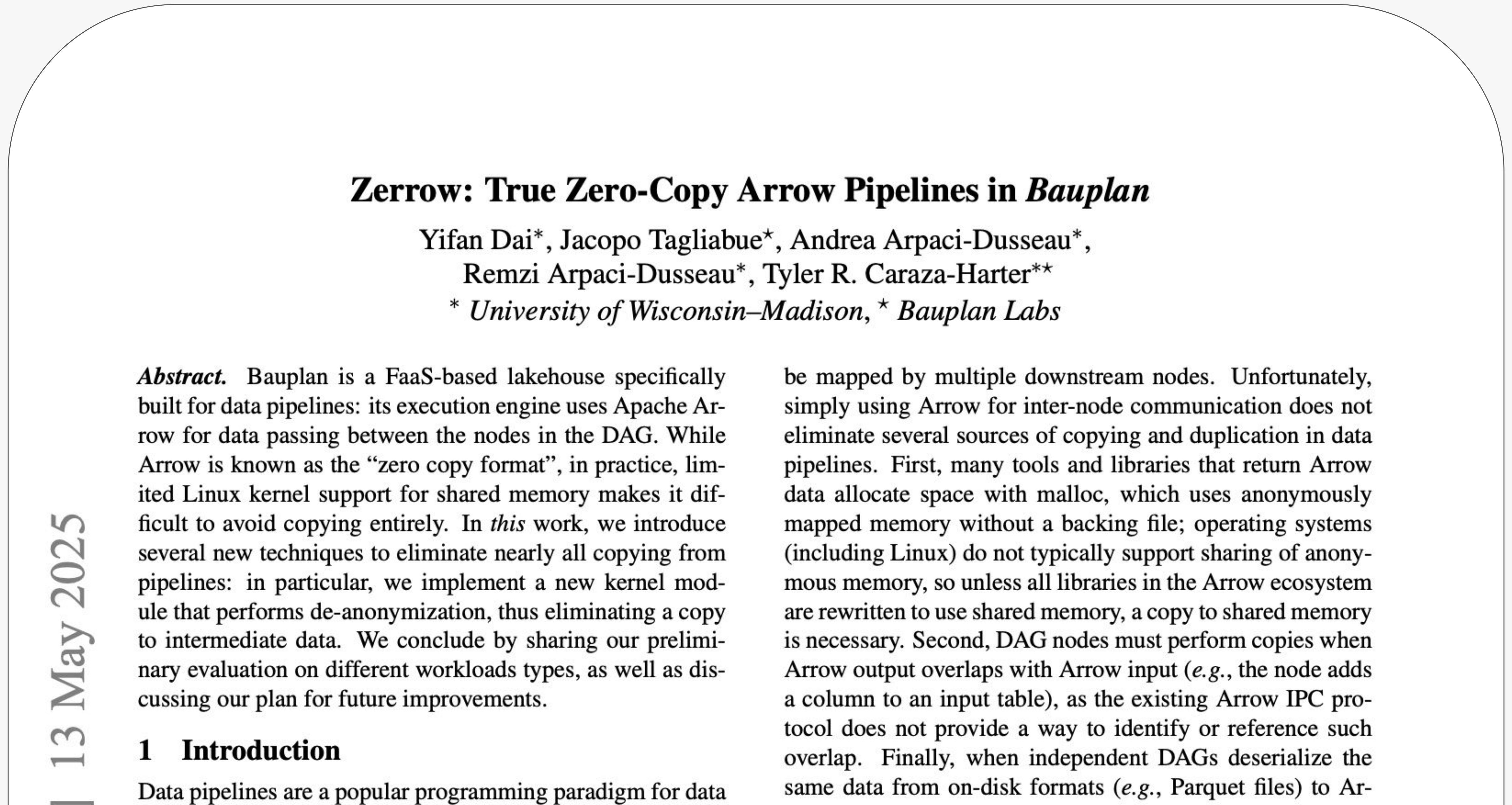
Table 3: Reading a dataframe from a parent (*c5.9xlarge*), avg. (SD) over 5 trials

	10M rows (6 GB)	*50M rows (30 GB)*
Parquet file in S3	1.26 (0.14)	6.14 (0.98)
Parquet file on SSD	0.92 (0.09)	4.37 (0.15)
Arrow Flight	0.96 (0.01)	4.69 (0.01)
Arrow IPC	**0.01 (0.00)**	**0.03 (0.01)**

IEEE BIG DATA 2024: Faas and Furious: abstraction and differential caching for efficient data pre-processing



RQ: *is* ***D*** *even feasible?*



Zerrow: True Zero-Copy Arrow Pipelines in *Bauplan*

Yifan Dai*, Jacopo Tagliabue*, Andrea Arpaci-Dusseau*,
Remzi Arpaci-Dusseau*, Tyler R. Caraza-Harter**
* *University of Wisconsin–Madison*, * *Bauplan Labs*

13 May 2025

Abstract. Bauplan is a FaaS-based lakehouse specifically built for data pipelines: its execution engine uses Apache Arrow for data passing between the nodes in the DAG. While Arrow is known as the "zero copy format", in practice, limited Linux kernel support for shared memory makes it difficult to avoid copying entirely. In *this* work, we introduce several new techniques to eliminate nearly all copying from pipelines: in particular, we implement a new kernel module that performs de-anonymization, thus eliminating a copy to intermediate data. We conclude by sharing our preliminary evaluation on different workloads types, as well as discussing our plan for future improvements.

1 Introduction

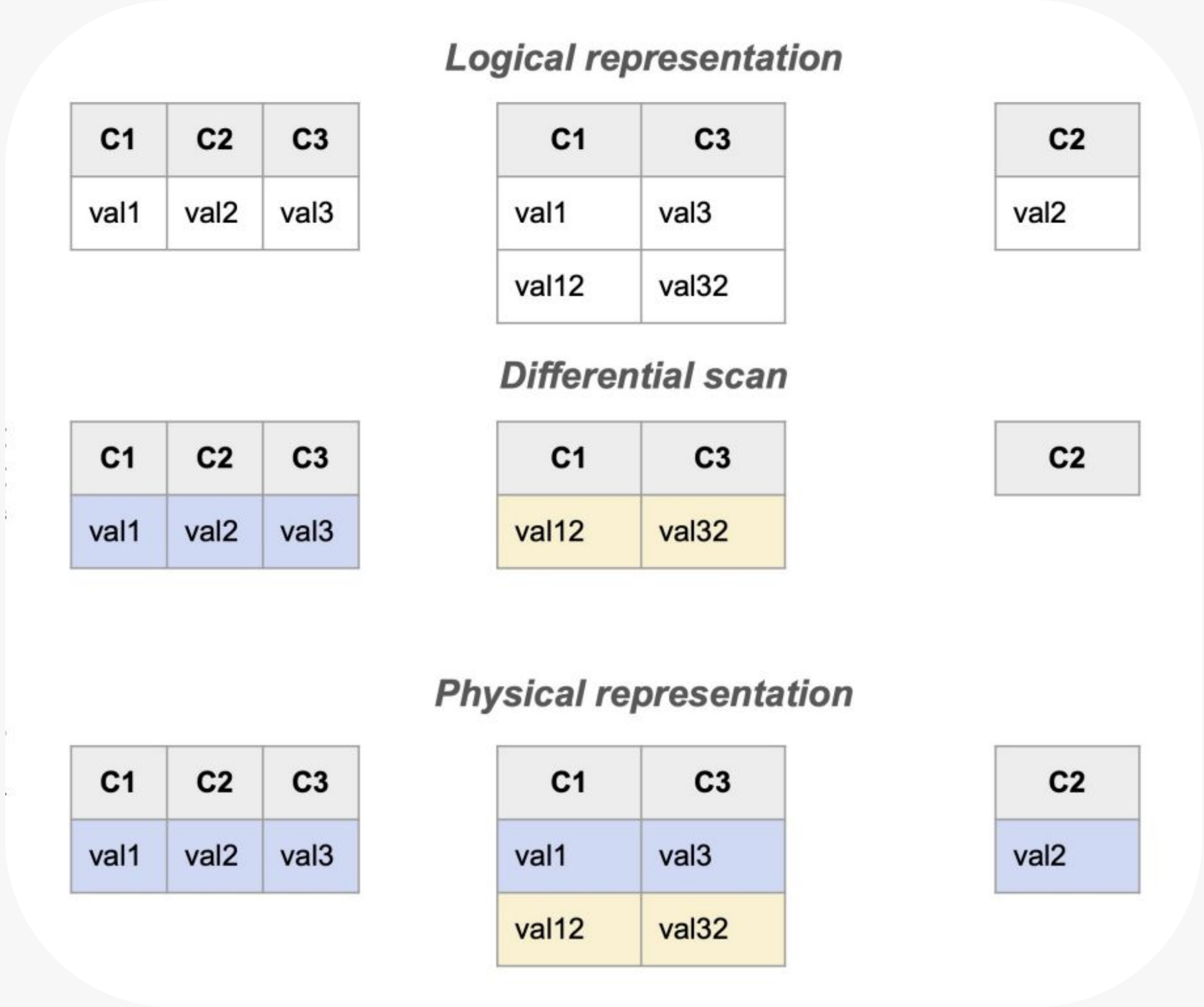
Data pipelines are a popular programming paradigm for data

be mapped by multiple downstream nodes. Unfortunately, simply using Arrow for inter-node communication does not eliminate several sources of copying and duplication in data pipelines. First, many tools and libraries that return Arrow data allocate space with malloc, which uses anonymously mapped memory without a backing file; operating systems (including Linux) do not typically support sharing of anonymous memory, so unless all libraries in the Arrow ecosystem are rewritten to use shared memory, a copy to shared memory is necessary. Second, DAG nodes must perform copies when Arrow output overlaps with Arrow input (*e.g.*, the node adds a column to an input table), as the existing Arrow IPC protocol does not provide a way to identify or reference such overlap. Finally, when independent DAGs deserialize the same data from on-disk formats (*e.g.*, Parquet files) to Ar-

Scans do not repeat themselves, but they often rhyme

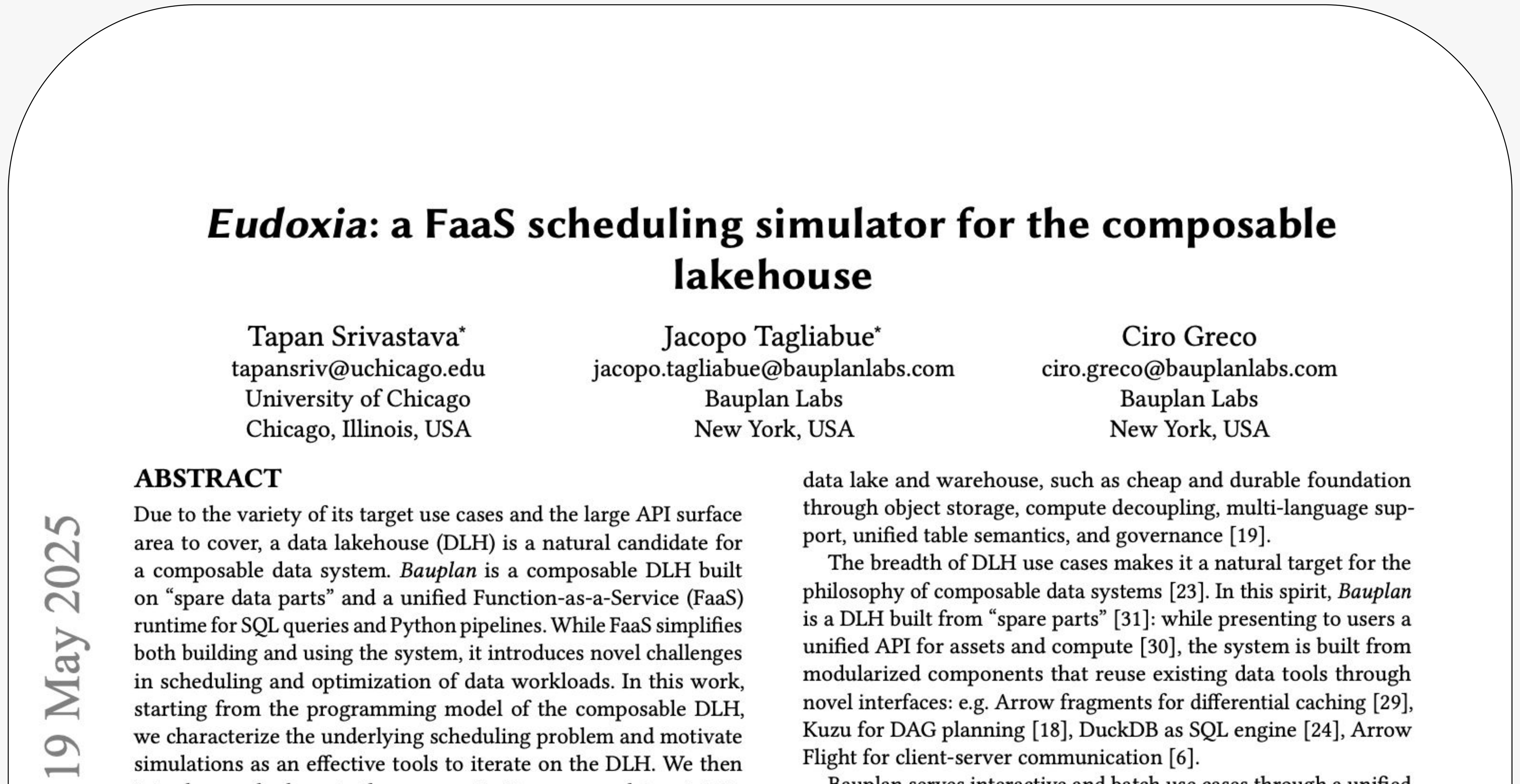
Differential cache:

- U1: “SELECT c1, c2, c3 FROM t WHERE eventTime BETWEEN 2023–01–01 AND 2023–02–01”
- U2: “SELECT c1, c3 … BETWEEN 2023–01–01 AND 2023–03–01”
- U1: “SELECT c2 … BETWEEN 2023–01–01 AND 2023–01–02”



IEEE BIG DATA 2024: Faas and Furious: abstraction and differential caching for efficient data pre-processing

RQ: *how do you manage concurrent functions?*



Eudoxia: a FaaS scheduling simulator for the composable lakehouse

Tapan Srivastava*
tapansriv@uchicago.edu
University of Chicago
Chicago, Illinois, USA

Jacopo Tagliabue*
jacopo.tagliabue@bauplanlabs.com
Bauplan Labs
New York, USA

Ciro Greco
ciro.greco@bauplanlabs.com
Bauplan Labs
New York, USA

19 May 2025

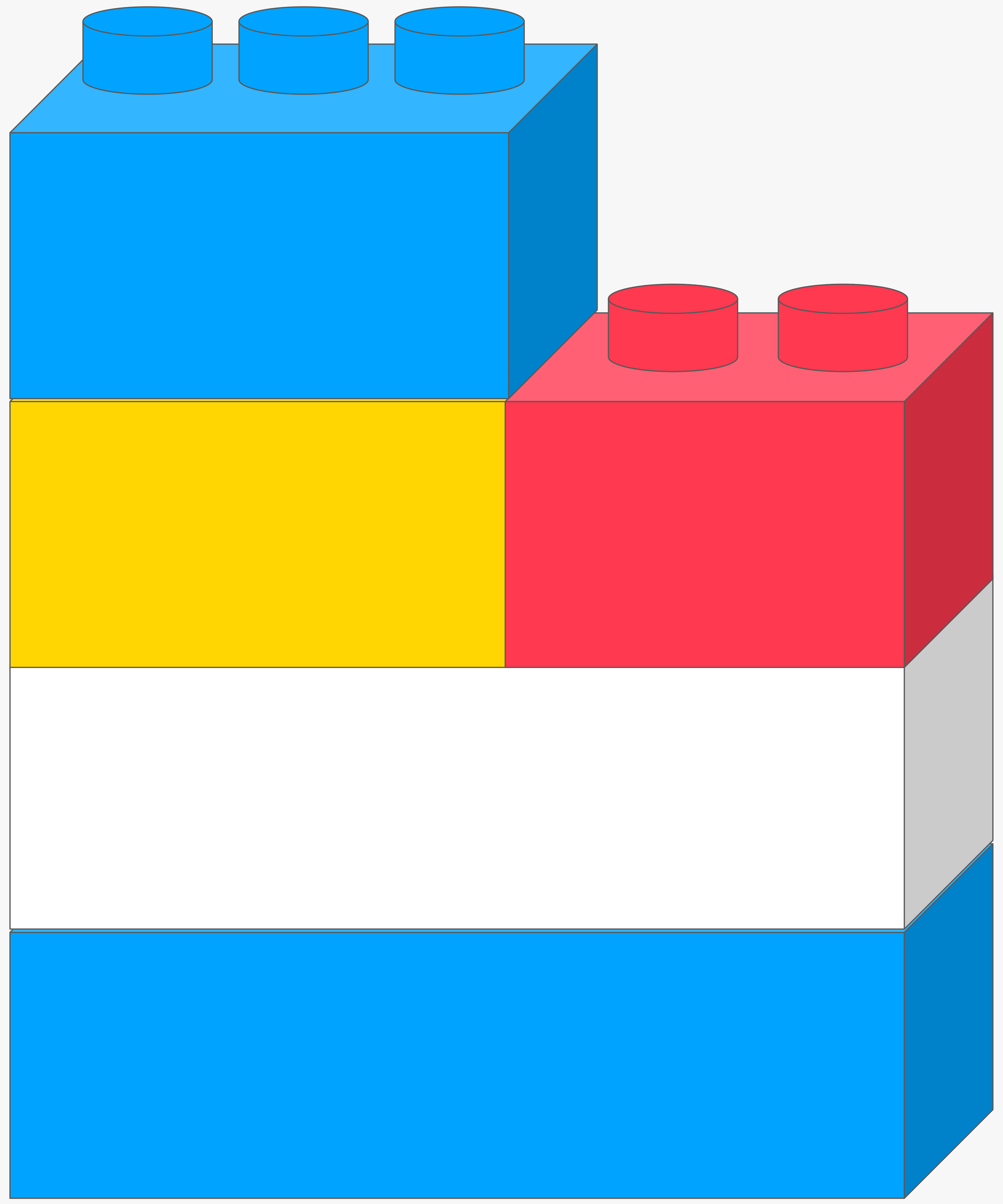
ABSTRACT

Due to the variety of its target use cases and the large API surface area to cover, a data lakehouse (DLH) is a natural candidate for a composable data system. *Bauplan* is a composable DLH built on "spare data parts" and a unified Function-as-a-Service (FaaS) runtime for SQL queries and Python pipelines. While FaaS simplifies both building and using the system, it introduces novel challenges in scheduling and optimization of data workloads. In this work, starting from the programming model of the composable DLH, we characterize the underlying scheduling problem and motivate simulations as an effective tools to iterate on the DLH. We then

data lake and warehouse, such as cheap and durable foundation through object storage, compute decoupling, multi-language support, unified table semantics, and governance [19].

The breadth of DLH use cases makes it a natural target for the philosophy of composable data systems [23]. In this spirit, *Bauplan* is a DLH built from "spare parts" [31]: while presenting to users a unified API for assets and compute [30], the system is built from modularized components that reuse existing data tools through novel interfaces: e.g. Arrow fragments for differential caching [29], Kuzu for DAG planning [18], DuckDB as SQL engine [24], Arrow Flight for client-server communication [6].

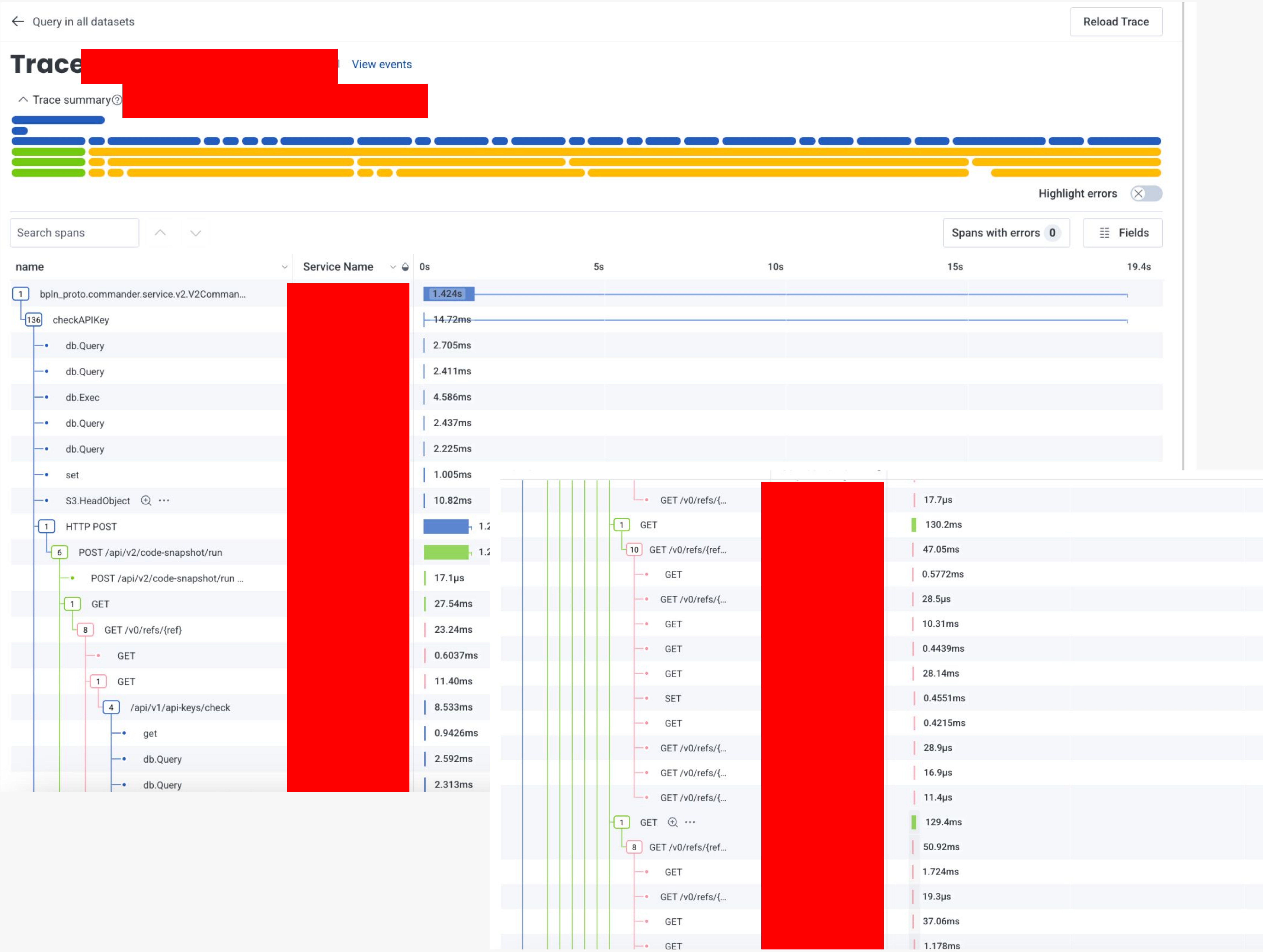
The Composable Lakehouse



composable

Low-level view of a `bauplan run`

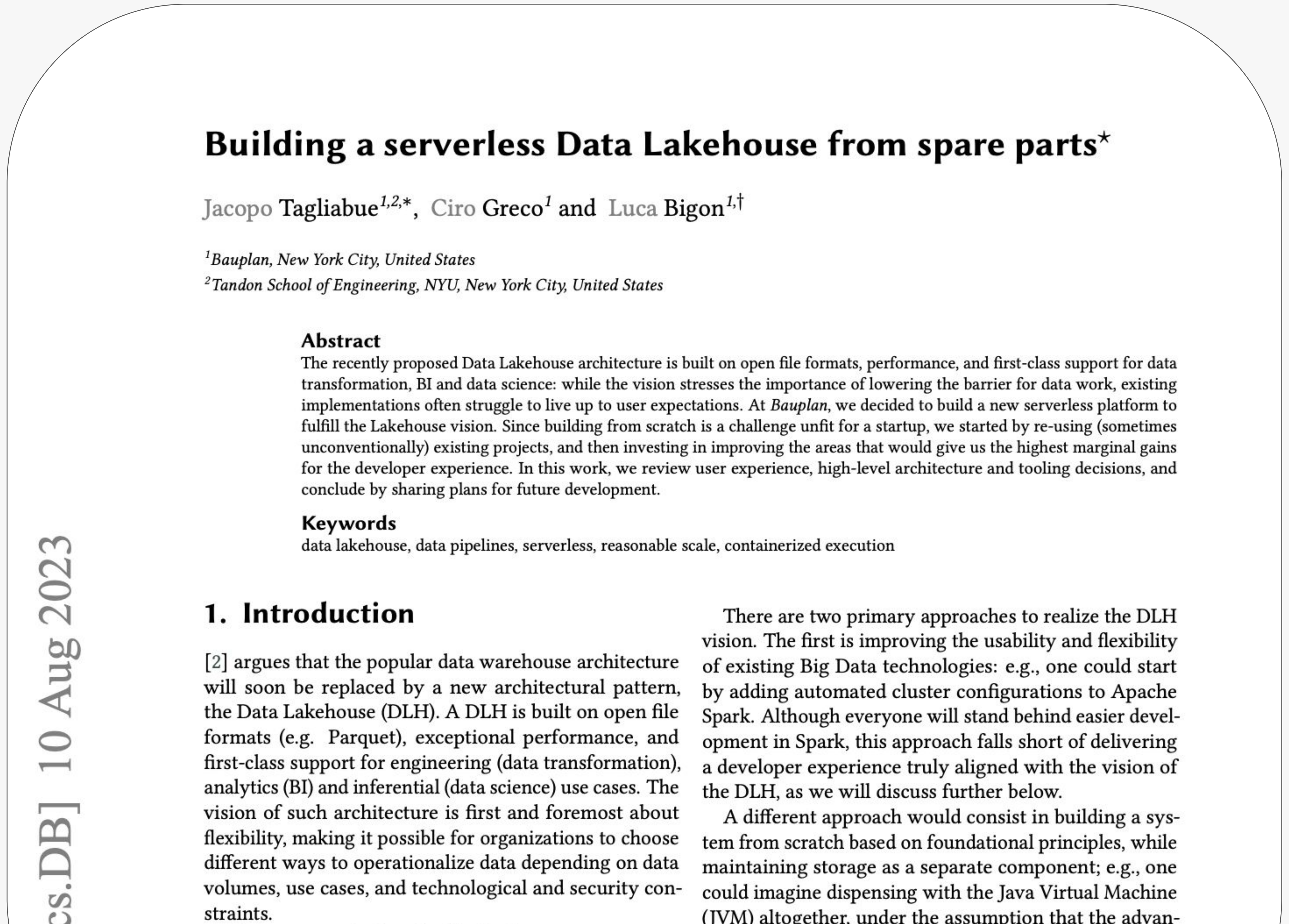
A single *run* spans hundreds of traces, across dozens of services, ranging from hyper-scaler PaaS to obscure open-source libraries.



The *composable* lakehouse

Bauplan is built around some core competencies plus "spare parts":

- FaaS runtime and abstractions are new
- Our query engine is a fork of DuckDb
- Our catalog is a fork of Nessie
- Our DAG planner is built on Kuzu
- Our Iceberg client is a fork of PyIceberg



Building a serverless Data Lakehouse from spare parts*

Jacopo Tagliabue[1,2,*], Ciro Greco[1] and Luca Bigon[1,†]

[1]Bauplan, New York City, United States
[2]Tandon School of Engineering, NYU, New York City, United States

Abstract
The recently proposed Data Lakehouse architecture is built on open file formats, performance, and first-class support for data transformation, BI and data science: while the vision stresses the importance of lowering the barrier for data work, existing implementations often struggle to live up to user expectations. At *Bauplan*, we decided to build a new serverless platform to fulfill the Lakehouse vision. Since building from scratch is a challenge unfit for a startup, we started by re-using (sometimes unconventionally) existing projects, and then investing in improving the areas that would give us the highest marginal gains for the developer experience. In this work, we review user experience, high-level architecture and tooling decisions, and conclude by sharing plans for future development.

Keywords
data lakehouse, data pipelines, serverless, reasonable scale, containerized execution

1. Introduction

[2] argues that the popular data warehouse architecture will soon be replaced by a new architectural pattern, the Data Lakehouse (DLH). A DLH is built on open file formats (e.g. Parquet), exceptional performance, and first-class support for engineering (data transformation), analytics (BI) and inferential (data science) use cases. The vision of such architecture is first and foremost about flexibility, making it possible for organizations to choose different ways to operationalize data depending on data volumes, use cases, and technological and security constraints.

There are two primary approaches to realize the DLH vision. The first is improving the usability and flexibility of existing Big Data technologies: e.g., one could start by adding automated cluster configurations to Apache Spark. Although everyone will stand behind easier development in Spark, this approach falls short of delivering a developer experience truly aligned with the vision of the DLH, as we will discuss further below.

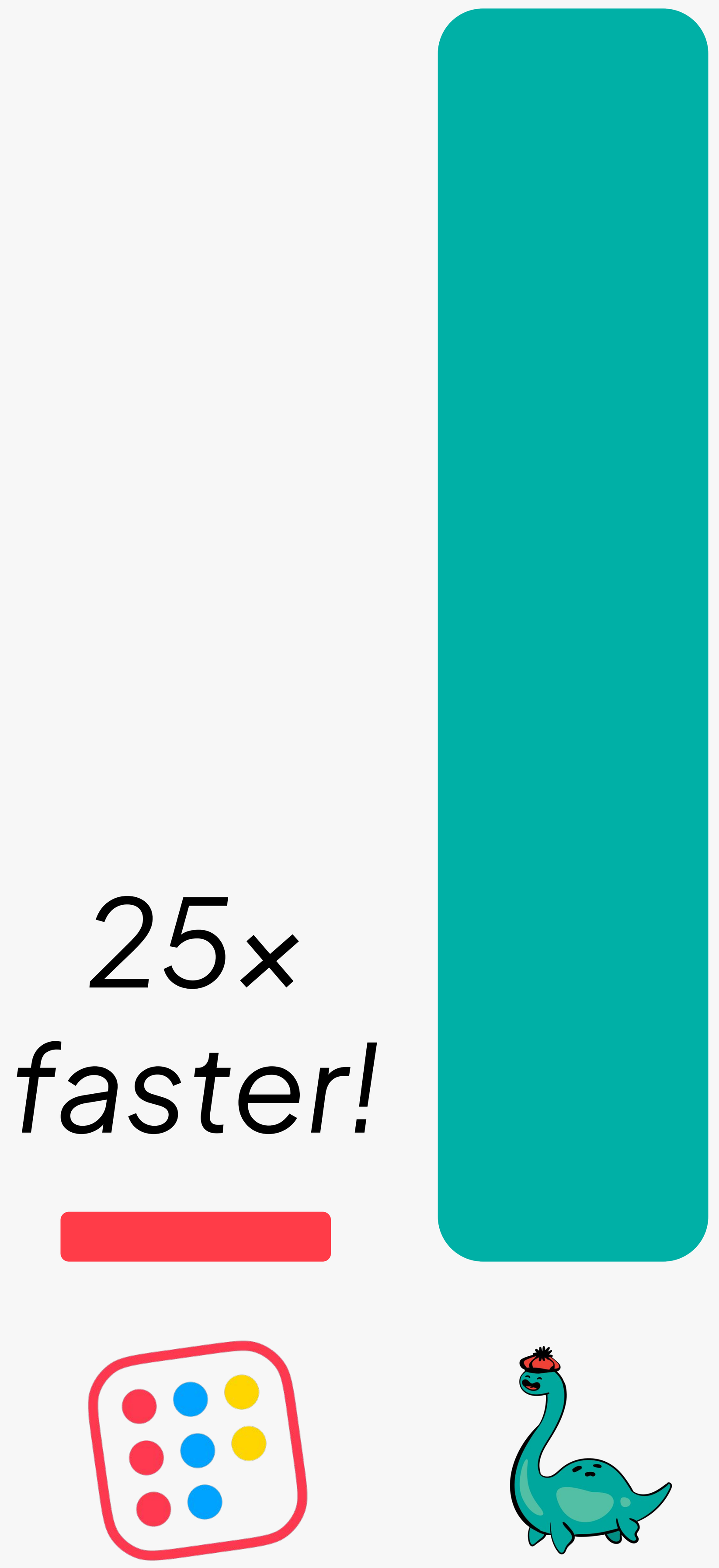
A different approach would consist in building a system from scratch based on foundational principles, while maintaining storage as a separate component; e.g., one could imagine dispensing with the Java Virtual Machine (JVM) altogether, under the assumption that the advan-

cs.DB] 10 Aug 2023

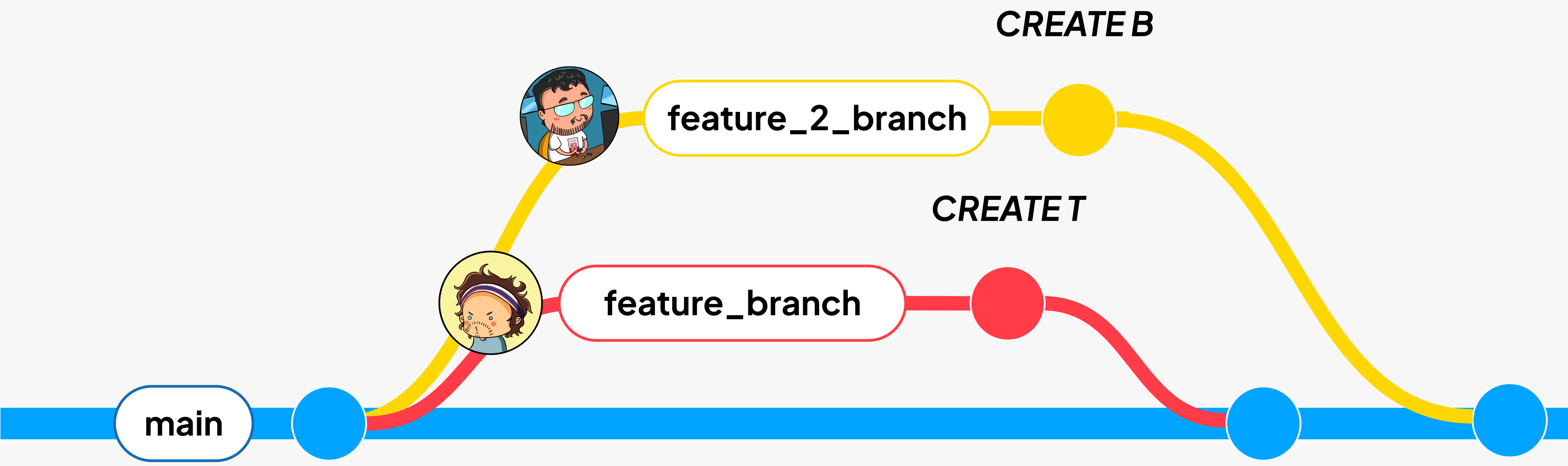
The *composable* lakehouse

We help develop libraries we use, improve them and often contribute back:

- PyIceberg
- Duckdb
- Nessie
- Datafusion
- Kuzu
- Iceberg-rust



How much “Git” is in Git-for-data?



engineering Jul 17, 2025 Written by Ciro Greco and Jacopo Tagliabue

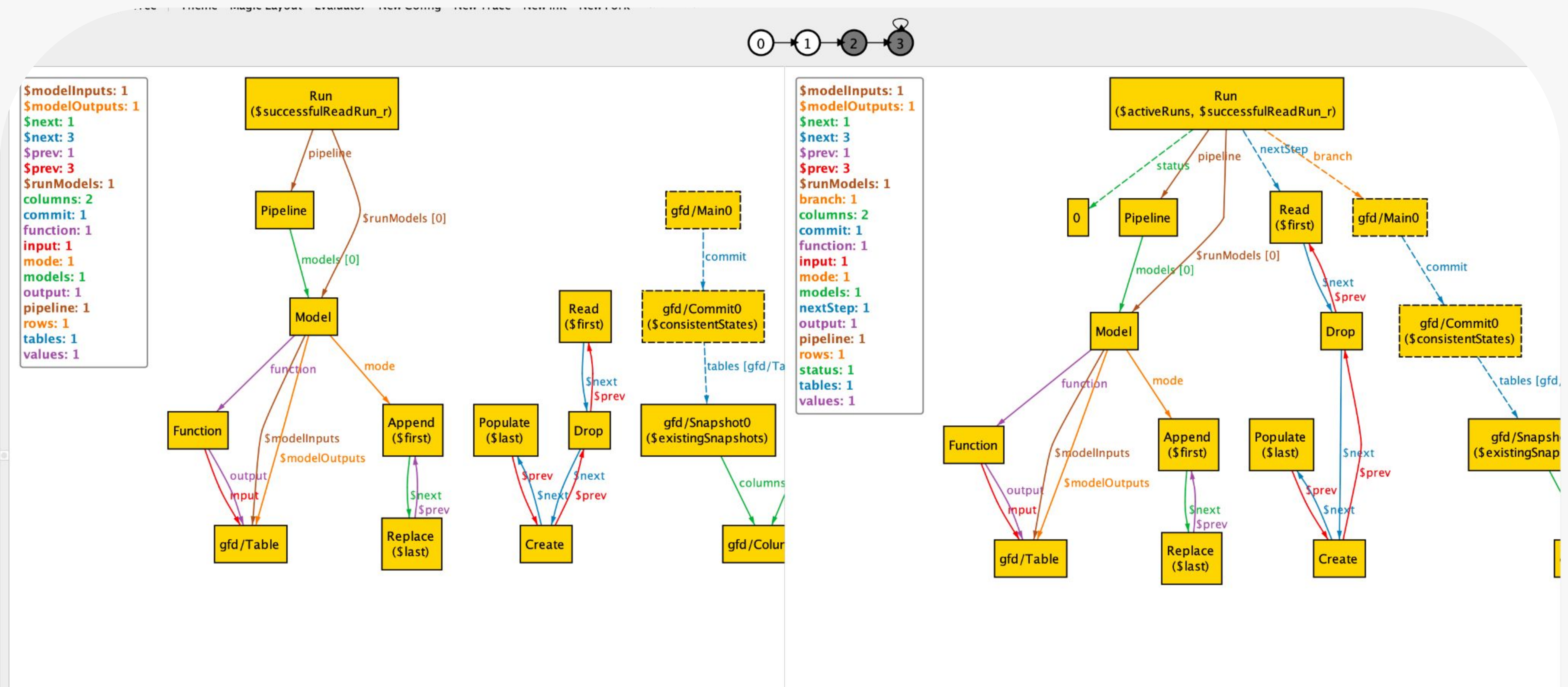
Git for Data: Formal Semantics of Branching, Merging, and Rollbacks (Part 1)

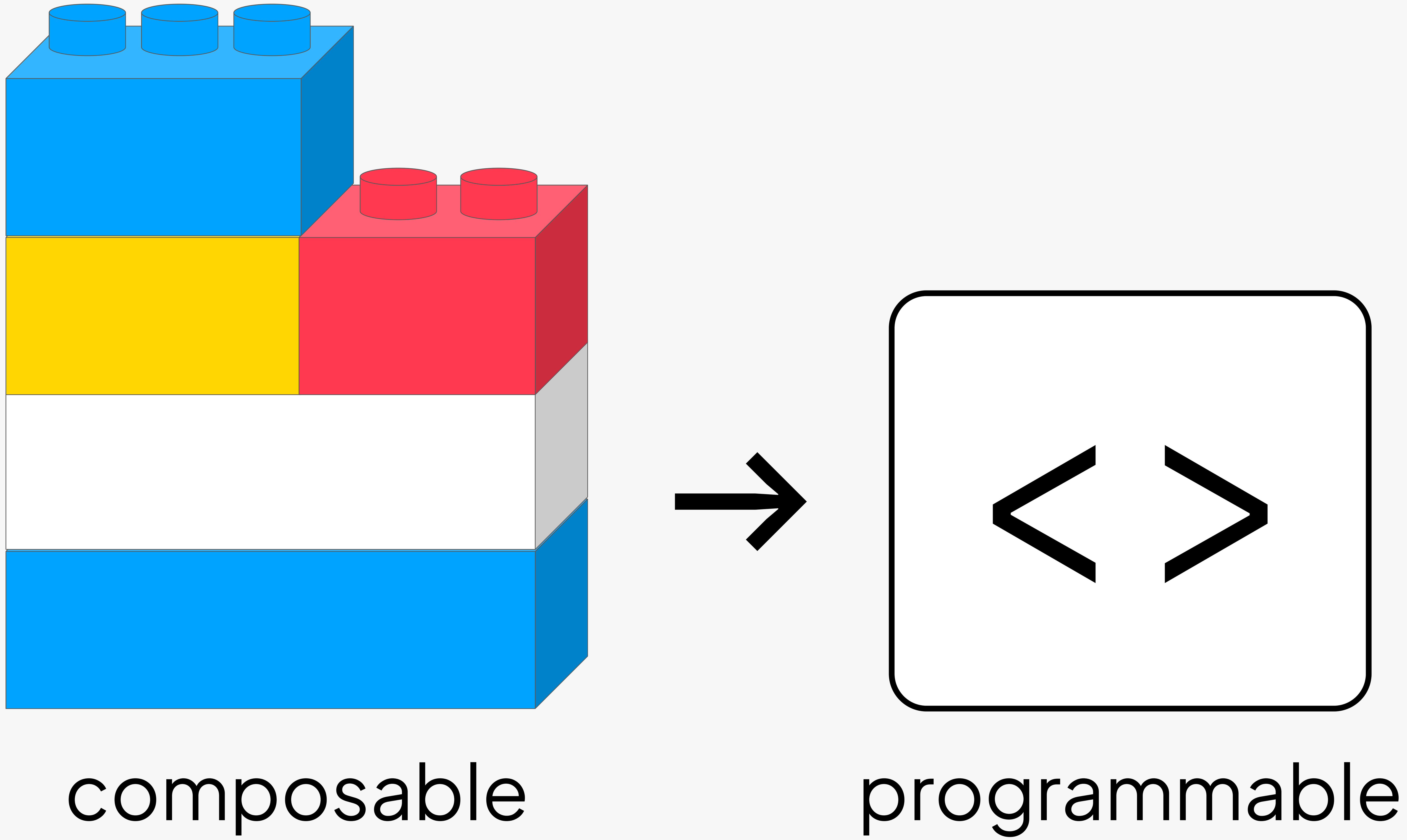
How formal methods help ensure safe, reproducible workflows in data lakehouses

How much "database" is in Git-for-data?

"We have discovered a truly marvelous proof of this, which this slide is too narrow to contain"

Lightweight formal models to verify data consistency in the face of failure, and run automated checks as we add new primitives!




composable
programmable

The *programmable* lakehouse

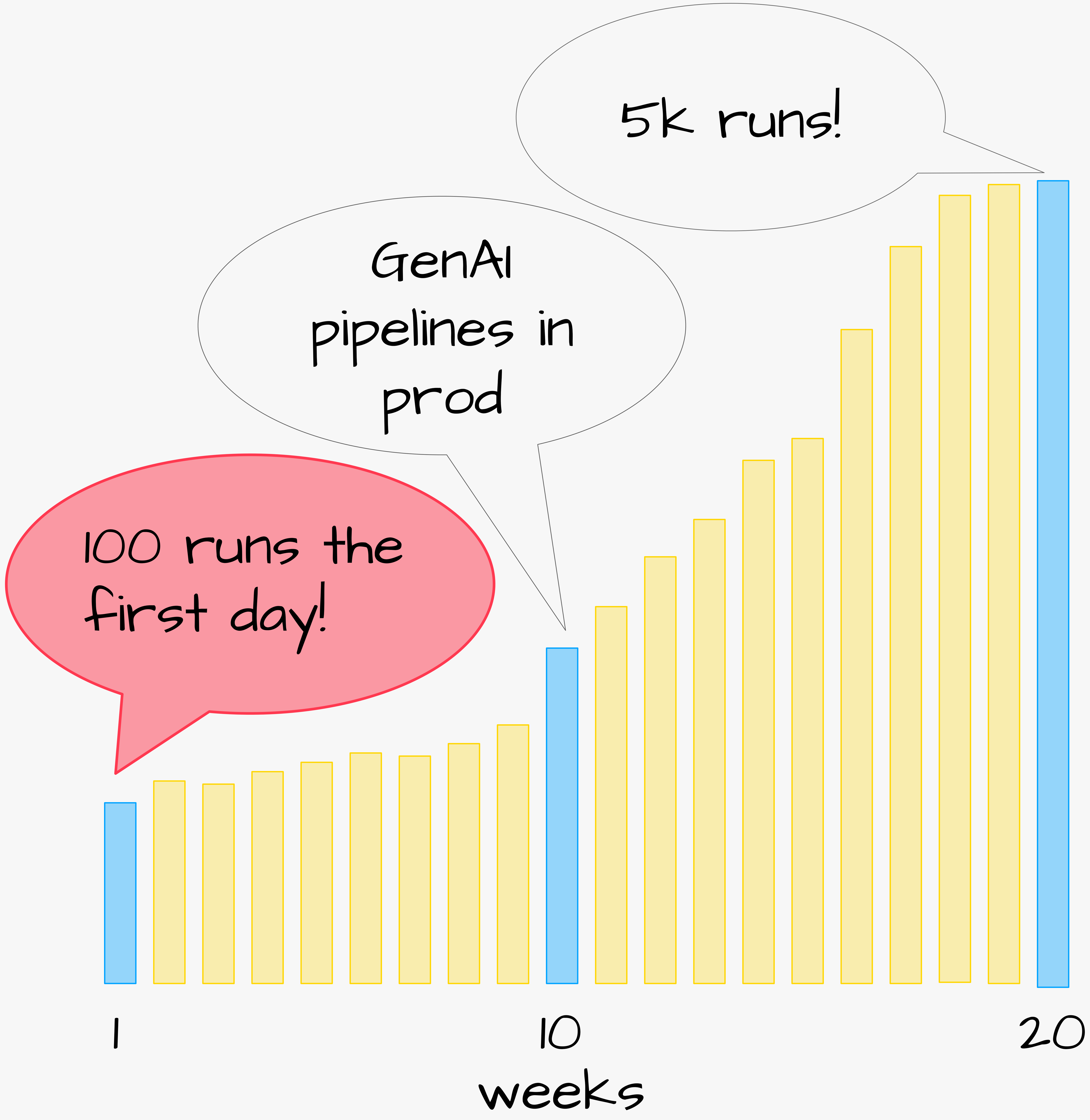
```python
import bauplan

@bauplan.model()
@bauplan.python(pip={'polars': '0.8.0'})
def augmented_dataset(
    input_table='nyc_taxi',
    columns=[col1,'col2']
    filter="datetime ='2022-12-15",
    model='?model'
):
    if model = 'chatgpt':
      # init the client here
    elif model = 'claude':
      # init the client here
    …
    return predictions
```

```python
import bauplan

client = bauplan.Client()
# run models on branches
for i, model in enumerate(models):
  model_branch = client.create_branch(
        branch=f"{i}_model",
        from_ref='main',
        params= { "model": model },
  )
  run_state = client.run(
        dir=my_pipeline,
        branch=agent_branch
  )

# merge the best version
client.merge_branch(
        source_ref=my_best_branch,
        into_branch='main'
  )
```

Inside composability != outside composability

The "API Ladder" philosophy

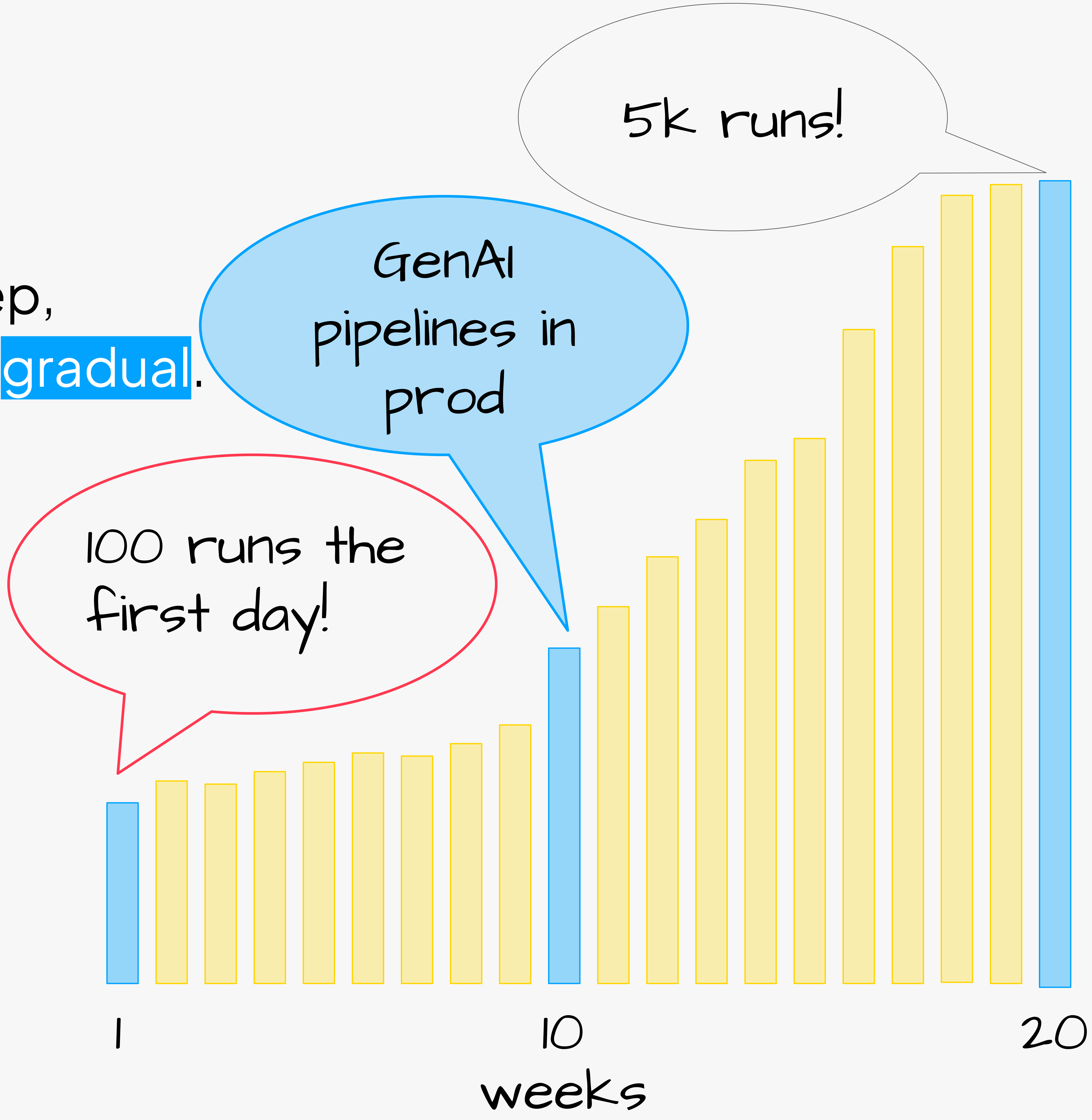
> In order to get started, beginners need an API to be convenient.



The "API Ladder" philosophy

> In order to take the next step, novices need the API to be gradual.

> In order to get started, beginners need an API to be convenient.

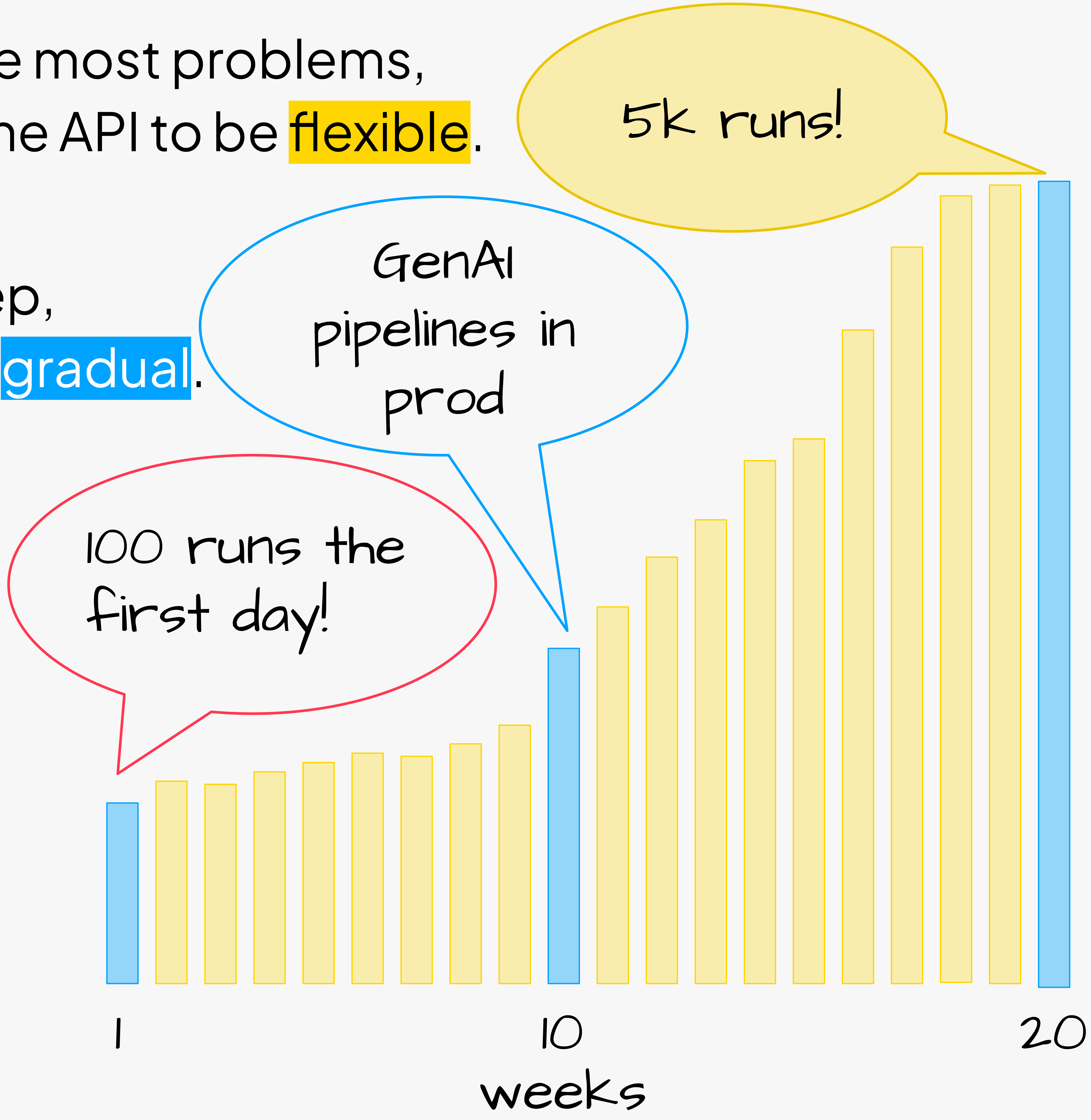


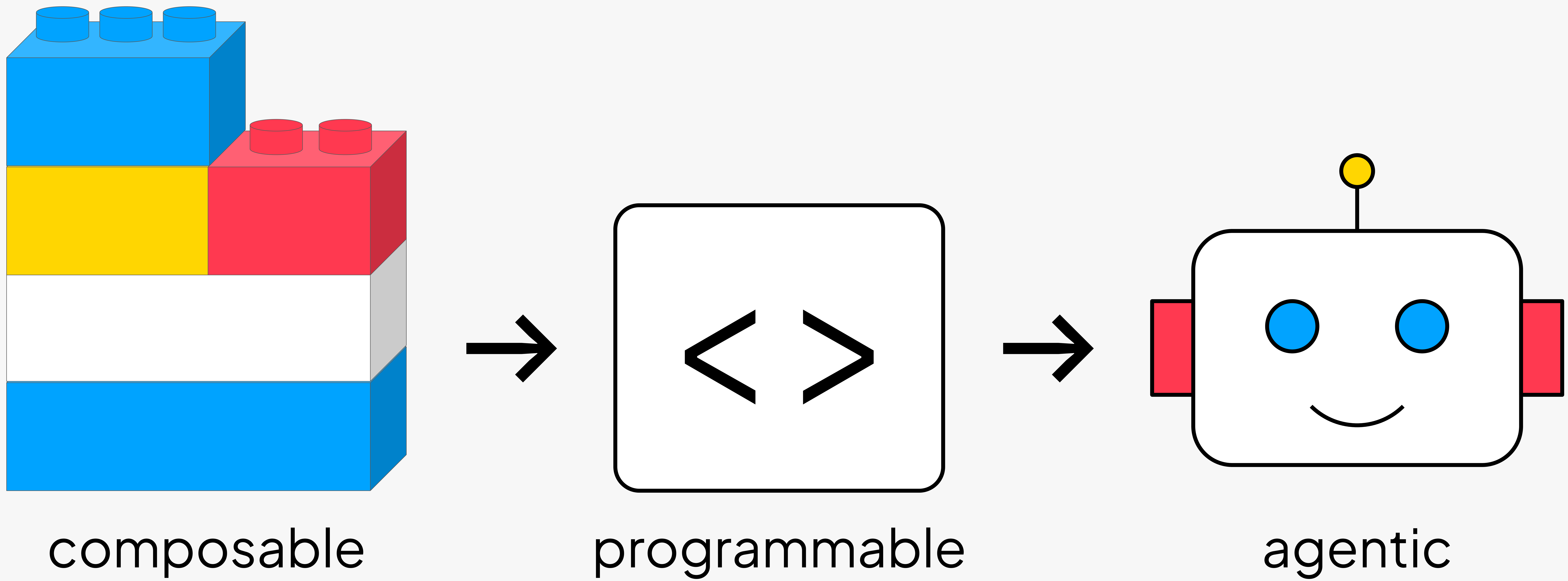
The "API Ladder" philosophy

| In order to solve most problems, experts need the API to be flexible.

| In order to take the next step, novices need the API to be gradual.

| In order to get started, beginners need an API to be convenient.




< >
composable
programmable
agentic

The *agentic* lakehouse

> **“Make something idiot-proof, and someone will come up with a better idiot”**

> Agents need easy-to-reason about APIs (check!), declarative infrastructure (check!) and the possibility of making mistakes without destroying downstream systems (check!).

> Bauplan APIs *are* the lakehouse: any model can run the full data life-cycle just with prompting!

Safe, Untrusted, “Proof-Carrying” AI Agents: towards the agentic lakehouse

Jacopo Tagliabue
Bauplan Labs
NYC, USA
jacopo.tagliabue@bauplanlabs.com

Federico Bianchi
TogetherAI
San Francisco, USA
federico@together.ai

Ciro Greco
Bauplan Labs
NYC, USA
ciro.greco@bauplanlabs.com

***Abstract*—Data lakehouses manage sensitive workloads where AI automation raises risks for trust, correctness, and governance. We argue that API-first, programmable lakehouses provide the right abstractions for *safe-by-design* agentic workflows. Using Bauplan as a case study, we show how data branching and declarative environments naturally extend to agents, enabling reproducibility and observability while reducing the attack surface. We present a proof-of-concept for repairing broken data pipelines, combining Bauplan, TogetherAI, and agentic loops with correctness checks inspired by proof-carrying code. Preliminary results demonstrate both the feasibility and challenges of untrusted AI agents operating safely on production data, outlining a path towards the agentic lakehouse.**

***Index Terms*—AI agents, lakehouse, data pipelines, versioning**

I. Introduction

The data lakehouse is the *de facto* cloud architecture for analytics and Artificial Intelligence (AI) workloads [2], [3], thanks to storage-compute decoupling, multi-language support

pipelines is a canary test for agent penetration in high-stake non-trivial scenarios, which are often hard for expert humans [10], [11]. We summarize our contributions as follows:

1) we model the data pipeline life-cycle in a next-gen programmable lakehouse, `Bauplan` [12]: our key perspective is that traditional lakehouses resist automation because APIs are an afterthought, with no attempt to serve heterogeneous use cases with a unified interface;
2) we review common objections to automation of high-stake workloads, in the light of the proposed abstractions for repairing data pipelines: in particular, we argue that our model promotes trustworthiness and correctness both in data and code artifacts;
3) we release working code[1], showing a proof of concept for self-repairing pipelines using `Bauplan`, `TogetherAI` and an agentic loop. We share tentative results from the prototype, provide preliminary analyses

We barely scratched the surface!

Want to know more?

2023

- CDMS@VLDB 2023

2024

- SIGMOD 2024
- MIDDLEWARE 2024 (with UMadison-Wisconsin)
- BIG DATA 2024

2025

- UNDER REVIEW 2025 (with UMadison-Wisconsin)
- CDMS@VLDB 2025 (with UChicago)

- **(Most) lakehouse** use cases can be served by a FaaS model
- **Composability** allows us to explore the design space quickly and cheaply

"It is not worth an intelligent man's time to be in the majority. By definition, there are already enough people to do that."

G.H. Hardy

jacopo.tagliabue@bauplanlabs.com

We are hiring!

bauplan